





# Custom Modules - Source Code

A number of custom code was generated to support this experiment.

## datasets.py

```
import io
from typing import Callable, List

import torch
from torch.utils.data.sampler import WeightedRandomSampler
from torchtext.data.utils import get_tokenizer
from torchtext.utils import unicode_csv_reader
from torchtext.vocab import Vocab

_default_tokenizer = get_tokenizer("basic_english")
DEFAULT_LABEL_TRANSFORM = lambda x: x
DEFAULT_TEXT_TRANSFORM = lambda x: _default_tokenizer(x)

def create_torch_data_loader(
    dataset: data.Dataset,
    vocabab: Vocab,
    label_transform: Callable = DEFAULT_LABEL_TRANSFORM,
    text_transform: Callable = DEFAULT_TEXT_TRANSFORM,
    weighted=True,
    **kwargs
):
    """Creates a Pytorch style data loader using a dataset and a precompiled vocab.

    The dataset returns "model-ready" data.

    Args:
        dataset: The raw text dataset to be used during inference
        vocabab: the prenade vocabulary used to index words/phrases
        label_transform: any operation used on the datasets label output
        text_transform: operation used on the raw text sentence outputs from the data
        weighted: whether to weight the samples based on class distribution
        **kwargs: any additional kwargs used by Pytorch DataLoaders.

    Returns:
        A PyTorch DataLoader to be used during training, eval, or test.
    """
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    def _collate_batch(batch):
        label_list, docid_list, text_list, offsets = [], [], [], [0]
        for (_label_, _docid_, _text_) in batch:
            label_list.append(label_transform(_label_))
            processed_text = torch.tensor(
                vocabab(text_transform(_text_)), dtype=torch.int64
            )
            text_list.append(processed_text)
            offsets.append(processed_text.size(0))
            docid_list.append(_docid_)
        label_list = torch.tensor(label_list, dtype=torch.int64)
        offsets = torch.tensor(offsets[:-1]).cumsum(dim=0)
        text_list = torch.cat(text_list)
        return label_list.to(device), text_list.to(device), offsets.to(device), docid_list

    if weighted:
        weights = dataset.sample_weights
        sampler = WeightedRandomSampler(weights=weights, num_samples=len(weights))
    else:
        sampler = None

    return data.DataLoader(
        dataset,
        collate_fn=_collate_batch,
        shuffle=(sampler is None),
        sampler=sampler,
        **kwargs
    )

class TSVRawTextIterableDataset(data.IterableDataset):
    """Dataset that loads TSV data incrementally as an iterable and returns raw text.

    This dataset must be traversed in order as it only reads from the TSV file as it is called.
    Unlabeled if the size of data is too large to load into memory at once.
    """
    def __init__(self, filepath: str, data_columns: List[int]):
        """Loads an iterator from a file.

        Args:
            filepath: location of the .tsv file
            data_columns: the columns in the .tsv that are used as feature data
        """
        self._number_of_items = _get_tsv_file_length(filepath)
        self._iterator = _create_data_from_tsv(
            filepath, data_column_indices=data_columns
        )
        self._current_position = 0

    def __iter__(self):
        return self

    def __next__(self):
        item = next(self._iterator)
        self._current_position += 1
        return item

    def __len__(self):
        return self._number_of_items

class TSVRawTextMapDataset(data.Dataset):
    """Dataset that loads all TSV data into memory and returns raw text.

    This dataset provides a map interface, allowing access to any entry.
    Useful for modifying the sampling or order during training.
    """
    def __init__(self, filepath: str, data_columns: List[int]):
        """Loads .tsv structured data into memory.

        Args:
            filepath: location of the .tsv file
            data_columns: the columns in the .tsv that are used as feature data
        """
        self._records = list(
            _create_data_from_tsv(filepath, data_column_indices=data_columns)
        )
        self._sample_weights, self._class_weights = self._calculate_weights()

    @property
    def sample_weights(self):
        return self._sample_weights

    @property
    def class_weights(self):
        return self._class_weights

    def _calculate_weights(self):
        targets = torch.tensor(
            [label if label > 0 else 0 for label, *_ in self._records]
        )
        unique, sample_counts = torch.unique(targets, return_counts=True)
        weight = 1.0 / sample_counts
        sample_weights = torch.tensor([weight[t] for t in targets])
        class_weights = weight / weight.sum()
        return sample_weights, class_weights

    def __getitem__(self, index):
        return self._records[index]

    def __len__(self):
        return len(self._records)

def _create_data_from_tsv(data_path, data_column_indices):
    with io.open(data_path, encoding="utf8") as f:
        reader = unicode_csv_reader(f, delimiter="\\t")
        for row in reader:
            data = [row[i] for i in data_column_indices]
            yield int(row[0]), row[1], "-".join(data)

def _get_tsv_file_length(data_path):
    with io.open(data_path, encoding="utf8") as f:
        row_count = sum(1 for row in f)

    return row_count
```

## models.py

```
import io
from typing import List
import numpy as np
import torch
from torchtext.data.utils import get_tokenizer, ngrams_iterator
from torchtext.utils import unicode_csv_reader
from torchtext.vocab import build_vocab_from_iterator
from torchtext.vocab import Vocab as vocab_builder
from torchtext.vocab import GloVe

def create_glove_with_unk_vector() -> GloVe:
    glove = GloVe()
    # Load the average vector for this glove embedding set to use for defaults.
    average_glove_vector = np.load("../datasets/glove_default_vector.npy")
    unk_init_vec = torch.from_numpy(average_glove_vector)
    # Extend the glove vectors with one for "unk"
    glove.vectors = torch.cat((glove.vectors, unk_init_vec.unsqueeze(0)))

    return glove

def create_vocab_from_glove(glove: GloVe):
    # Since glove is already ordered and not a counter, we overload the
    # constructor to align the indices.
    unk_token = "cunk"
    vocab = vocab_builder(glove.stoi, min_freq=0)
    vocab.append_token(unk_token)
    vocab.set_default_index(vocab[unk_token])

    return vocab

def create_vocab_from_tsv(
    filepath: str,
    column_indices_to_use: List[int],
    minimum_word_freq: int = 1,
    ngrams: int = 1,
):
    """Creates a PyTorch vocab object from a TSV file.

    The resulting vocab object converts words to indices for assisting in embedding and DL operations.

    Args:
        filepath: The location of the TSV file
        minimum_word_freq: How many times a word must appear to be included
        ngrams: The size of ngrams to use for the vocab
        column_indices_to_use: Which columns from the TSV are part of the actual feature set

    Returns:
        A torchtext vocab object.
    """
    unk_token = "cunk"
    vocab = build_vocab_from_iterator(
        _tsv_iterator(filepath, ngrams=ngrams, column_indices=column_indices_to_use,
            min_freq=minimum_word_freq,
            specials=[unk_token],
        )
    )
    vocab.set_default_index(vocab[unk_token])
    return vocab

def _tsv_iterator(data_path, ngrams, column_indices):
    # Spacy has novel tokenizer
    tokenizer = get_tokenizer("basic_english")
    with io.open(data_path, encoding="utf8") as f:
        reader = unicode_csv_reader(f, delimiter="\\t")
        for row in reader:
            row_iter = [row[i] for i in column_indices]
            tokens = " ".join(row_iter)
            yield ngrams_iterator(tokenizer(tokens), ngrams)

train.py
from collections import Counter
from logging import log
from typing import Any, Callable, Dict, Tuple

import torch
import torch.nn as nn
from sklearn.metrics import precision_recall_fscore_support, average_precision_score
from torch.utils import data
from torch.nn.utils.tensorboard import SummaryWriter
from tqdm import tqdm

def predict(model: nn.Module, text: torch.Tensor) -> int:
    """Predicts the class of a specific text given converted features.

    Args:
        model: the model to use for prediction/inference
        text: the previously converted text (using the prior dictionary)

    Returns:
        The predicted label for the provided text.
    """
    model.eval()
    no_offset = torch.tensor([0])
    with torch.no_grad():
        pred_scores = model(text, no_offset)
        pred_label = pred_scores.argmax(1).item()
        return pred_label

def train_epoch(
    epoch_num: int,
    model: nn.Module,
    optimizer: torch.optim.Optimizer,
    loss_function: Callable,
    dataloader: data.DataLoader,
    start_iter: int = 0,
    log_interval: int = 100,
    writer: SummaryWriter = None,
) -> int:
    """Performs training on a single pass through a dataloader.

    Args:
        epoch_num: The current number of this epoch of training.
        model: The PyTorch module to train
        optimizer: The optimizer to use for training.
        loss_function: The function that calculates loss between truth and prediction.
        dataloader: Provides a properly formatted batch of data at each iteration.
        start_iter: The iteration this epoch started on. Used for plotting.
        log_interval: How often to log the scores.
        writer: Tensorboard summary writer.

    Returns:
        The value of the start_iter plus number of batches performed this epoch.
    """
    batch_counter = start_iter
    model.train()
    with tqdm(dataloader, unit="batch", bar_format="{desc:>20}{percentage:3.0f}%{bar}|{r_bar}") as teepoch:
        for batch in teepoch:
            batch_counter += 1
            teepoch.set_description(f"Epoch {epoch_num}")
            results = train_step(batch, model, optimizer, loss_function)
            teepoch.set_postfix(loss=results["loss"], accuracy=results["accuracy"])

            if writer is not None and batch_counter % log_interval == 0:
                writer.add_scalars("training", results, batch_counter)

    return batch_counter

def train_step(
    batch: Tuple[torch.Tensor, ...],
    model: nn.Module,
    optimizer: torch.optim.Optimizer,
    loss_function: Callable,
) -> Dict[str, float]:
    """Performs a single training step on a model.

    Args:
        batch: A previously formatted batch of data.
        model: Torch model to perform training on.
        optimizer: The optimizer class used in training
        loss_function: The callable function to generate loss between prediction and truth.

    Returns:
        The different metrics generated this training step.
    """
    labels, text, offsets, *_ = batch

    optimizer.zero_grad()
    predicted_scores = model(text, offsets)
    loss = loss_function(predicted_scores, labels)
    loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), 0.1)
    optimizer.step()

    predicted_labels = predicted_scores.argmax(1)

    accuracy = (predicted_labels == labels).sum().item() / labels.size(0)

    y_true = labels.detach().cpu().numpy()
    y_pred = predicted_labels.cpu().numpy()
    precision, recall, fscore, support = precision_recall_fscore_support(
        y_true,
        y_pred,
        average="binary",
        zero_division=0,
    )

    results = {
        "loss": loss.detach().cpu().item(),
        "accuracy": accuracy,
        "precision": precision,
        "recall": recall,
        "fscore": fscore,
    }
    return results

def evaluate_epoch(
    epoch_num: int,
    model: nn.Module,
    loss_function: Callable,
    dataloader: data.DataLoader,
    writer: SummaryWriter = None,
) -> Dict[str, float]:
    """Performs validation on a single pass through a dataloader.

    Args:
        epoch_num: The current number of this epoch of training.
        model: The PyTorch module to train
        loss_function: The function that calculates loss between truth and prediction.
        dataloader: Provides a properly formatted batch of data at each iteration.
        writer: Tensorboard summary writer.

    Returns:
        The average validation metrics for the whole dataset.
    """
    model.eval()
    aggregate_results = Counter()
    with tqdm(dataloader, unit="batch", bar_format="{desc:>20}{percentage:3.0f}%{bar}|{r_bar}") as tepoch:
        for batch in tepoch:
            tepoch.set_description(f"Validation: {epoch_num}")
            results = evaluate_step(batch, model, loss_function)
            tepoch.set_postfix(loss=results["loss"], accuracy=results["accuracy"])
            aggregate_results += Counter(results)

    average_results = {
        key: aggregate_results[key] / tepoch.total for key in aggregate_results
    }
    writer.add_scalars("validation", average_results, epoch_num)
    return average_results

def evaluate_step(
    batch: Tuple[torch.Tensor, ...],
    model: nn.Module,
    loss_function: Callable,
) -> Dict[str, float]:
    """Performs a single validation step on a model.

    Args:
        batch: A previously formatted batch of data.
        model: Torch model to perform training on.
        loss_function: The callable function to generate loss between prediction and truth.

    Returns:
        The different metrics generated this training step.
    """
    labels, text, offsets, *_ = batch

    with torch.no_grad():
        predicted_scores = model(text, offsets)
        loss = loss_function(predicted_scores, labels)
        predicted_labels = predicted_scores.argmax(1)
        accuracy = (predicted_labels == labels).sum().item() / labels.size(0)

        y_true = labels.detach().cpu().numpy()
        y_pred = predicted_labels.cpu().numpy()

        precision, recall, fscore, support = precision_recall_fscore_support(
            y_true,
            y_pred,
            average="binary",
            zero_division=0,
        )

    results = {
        "loss": loss.detach().cpu().item(),
        "accuracy": accuracy,
        "precision": precision,
        "recall": recall,
        "fscore": fscore,
    }
    return results

vocab.py
import io
from typing import List
import numpy as np
import torch
from torchtext.data.utils import get_tokenizer, ngrams_iterator
from torchtext.utils import unicode_csv_reader
from torchtext.vocab import build_vocab_from_iterator
from torchtext.vocab import Vocab as vocab_builder
from torchtext.vocab import GloVe

def create_glove_with_unk_vector() -> GloVe:
    glove = GloVe()
    # Load the average vector for this glove embedding set to use for defaults.
    average_glove_vector = np.load("../datasets/glove_default_vector.npy")
    unk_init_vec = torch.from_numpy(average_glove_vector)
    # Extend the glove vectors with one for "unk"
    glove.vectors = torch.cat((glove.vectors, unk_init_vec.unsqueeze(0)))

    return glove

def create_vocab_from_glove(glove: GloVe):
    # Since glove is already ordered and not a counter, we overload the
    # constructor to align the indices.
    unk_token = "cunk"
    vocab = vocab_builder(glove.stoi, min_freq=0)
    vocab.append_token(unk_token)
    vocab.set_default_index(vocab[unk_token])

    return vocab

def create_vocab_from_tsv(
    filepath: str,
    column_indices_to_use: List[int],
    minimum_word_freq: int = 1,
    ngrams: int = 1,
):
    """Creates a PyTorch vocab object from a TSV file.

    The resulting vocab object converts words to indices for assisting in embedding and DL operations.

    Args:
        filepath: The location of the TSV file
        minimum_word_freq: How many times a word must appear to be included
        ngrams: The size of ngrams to use for the vocab
        column_indices_to_use: Which columns from the TSV are part of the actual feature set

    Returns:
        A torchtext vocab object.
    """
    unk_token = "cunk"
    vocab = build_vocab_from_iterator(
        _tsv_iterator(filepath, ngrams=ngrams, column_indices=column_indices_to_use,
            min_freq=minimum_word_freq,
            specials=[unk_token],
        )
    )
    vocab.set_default_index(vocab[unk_token])
    return vocab

def _tsv_iterator(data_path, ngrams, column_indices):
    # Spacy has novel tokenizer
    tokenizer = get_tokenizer("basic_english")
    with io.open(data_path, encoding="utf8") as f:
        reader = unicode_csv_reader(f, delimiter="\\t")
        for row in reader:
            row_iter = [row[i] for i in column_indices]
            tokens = " ".join(row_iter)
            yield ngrams_iterator(tokenizer(tokens), ngrams)
```